Smart Executives for Autonomous Spacecraft

(To appear in *IEEE Intelligent Systems*, October 1998)

Erann Gat
Jet Propulsion Laboratory
California Institute of Technology

Barney Pell
Research Institute for Advanced Computer Science
NASA Ames Research Center

1. Introduction

Interplanetary spacecraft are traditionally controlled by sequences of commands generated by a team of engineers. These sequences are uplinked to the spacecraft, which executes them in a more-or-less linear fashion. The commands that constitute a sequence range from very low-level commands (e.g. "Toggle power switch 27.") to relatively high-level commands such as "Turn the spacecraft to attitude A."

Sequence-based commanding has sucessfully met the challenge of controlling spacecraft many hundreds of millions of miles from Earth with extreme precision and often in the face of hardware failure. But the manual generation of sequences is difficult, time-consuming, and expensive. This is mainly because sequences are executed open-loop, so erroneous sequences can cause loss of data or, in extreme cases, loss of an entire mission. Thus, sequences must be constructed with exquisite care and painstakingly checked and re-checked for correctness before they are uplinked to the spacecraft. Furthermore, any unexpected hardware failures or environmental interactions during sequence execution will cause the state of the spacecraft to diverge from its expected state. This usually causes the spacecraft to abort the sequence and enter a safe mode, requiring additional time-consuming and expensive human intervention.

Efforts are currently underway at the Jet Propulsion Laboratory (JPL) and NASA Ames Research Center (ARC) to design a **control system** for a new generation of spacecraft that can operate with much less human intervention, and execute missions in unmodelled environments such as the surfaces of comets. These **autonomous spacecraft** are designed to accept very high level commands and execute them reliably even in the face of hardware failure, unexpected environmental interactions, and errors or inconsistencies in the command set. To accomplish this requires radical rethinking of the entire spacecraft command process and control architecture. For example, some **planning functions** that are currently performed on the ground by humans must now be performed automatically by the spacecraft. Data representations and processes designed for humans in the loop must be redesigned to make them suitable for **on-board operation**.

In this article we explore the design of an executive for an autonomous spacecraft. The executive is responsible for translating high-level commands, whether they come from the ground or from an on-board planner, into the

low-level commands understood directly by the spacecraft hardware. This requires a paradigm shift from an open-loop sequence-based executive to a closed-loop executive that is aware of and knows how to respond robustly to unexpected contingencies.

To understand the operation of a closed-loop autonomous spacecraft executive it is instructive to begin with a description of how traditional sequence-based commands work. This will illustrate some of the complex and subtle issues that an autonomous executive must face.

2. Spacecraft Commanding 101

To get a feel for some of the issues involved in generating and analyzing spacecraft command sequences, consider a very simple command: turning on the power for a device. Electrical power is a heavily oversubscribed resource on a spacecraft, and so if one is not careful a power-on command can be executed in a situation where there is not enough power available on the power bus to operate the device. The consequences of such a mistake can be catastrophic.

Overloading a power bus causes the voltage on the bus to drop below the level at which the devices on the bus can operate reliably, a condition known as a "bus undervolt." On an unmanned spacecraft, this triggers emergency hardware interlocks that turn off the power to all but the most vital subsystems, an event known as a "bus trip." After a bus trip, the spacecraft goes into a mode of operation known as "safe mode" where it does nothing but try to establish contact with Earth. Getting the spacecraft out of safe mode requires manual intervention, which takes hours if not days. If a spacecraft enters safe mode just before a crucial science observation the opportunity for that observation is usually irretrivably lost. If a spacecraft enters safe mode during a critical mission maneuver such as an orbit insertion the entire mission can be lost. So even a simple thing like turning on a power switch is potentially catasrophic[1].

The situation is further complicated by the fact that there is usually no way to measure the amount of power available on a spacecraft. Sequences are designed based on predictive models of available power, which is a complicated function of the spacecraft state, which in turn is a function of the sequence being executed. Thus, every command in a sequence can interact with every other command in the sequence. Making a change in one part of the sequence can cause a catastrophic failure in a different part of the sequence. The situation is exacerbated by the fact that spacecraft resource margins are always very small. As a result, sequences are very brittle; the tiniest change can have a global and potentially catastrophic impact. This is what makes sequences so difficult and expensive to develop. (Critical sequences such as orbit insertions, which can cause loss of mission if they enter safe mode, can take many years to generate.)

---

[1]On the Cassini spacecraft, if certain non-critical devices fail during Saturn orbit insertion the sequence will proceed without them despite the fact that backups are available. Because of the possibility of a bus trip, the risk to the mission of turning on the backup is deemed to be higher then the risk of proceeding without the device.

Of course, a bus trip is not the only thing that can go wrong. Turning on a device is itself a fairly complicated process that involves many intermediary devices, any one of which can fail. A device's power terminal is typically connected to a power distribution unit (PDU) which contains a number of power switches that control the flow of electricity from a power bus to various spacecraft devices. (See figure 1.) The PDU is in turn connected to a peripheral data communications bus which is also connected to the spacecraft's computer through a bus controller (which controls a data communications bus, not to be confused with the power bus). Any of these components may be redundant, including the computer.

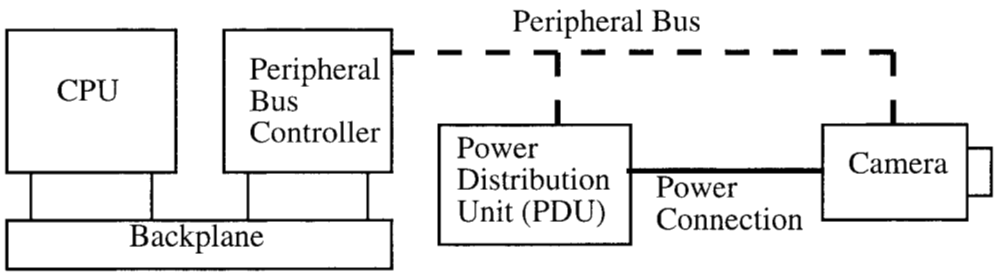

Figure 1: A simplified block diagram of the hardware components involved in turning on a camera.

The code that implements the sequence command, "Turn on device X," implements something along the lines of:

> Ask the bus controller to send a message to the PDU to change the state of the device X power switch to ON.

So an anomaly in turning on the power to a device could be caused by a problem in the device itself, or it could result from a problem in the PDU, the PDU's bus interface, the bus master, or the power source. It could also be caused by some other device on the spacecraft. For example, a bus client can fail in a way that causes it to inject noise on the bus and interfere with the operation of other bus clients. In fact, just about anything on the spacecraft can cause anomalous behavior anywhere else on the spacecraft. Unexpected effects can also be caused by spacecraft position or attitude, or interactions with external environmental phenomena like radiation.

An example of the kinds of interactions that can occur: on Cassini, firing the main engine can leave chemical deposits on the lens of the science camera, causing it to malfunction. To correct this problem, the camera is equipped with a little windshield wiper to clean the lens. When this wiper is operating, the effects of the reaction torque can be measured in the spacecraft's overall attitude.

3. Procedural and Declarative Knowledge Representations

By the time a command like "Turn on device X" is incorporated into a sequence a tremendous amount of knowledge has been brought to bear, including knowledge of mission goals, the design of the spacecraft, physics, and common sense. This knowledge is all "compiled away" in the final

sequence. There is no way to extract from a sequence the reasoning that led to its construction, or the constraints that it was designed to obey.

For example, consider the simple sequence, "Turn on the camera, then take a picture." We humans know that the order in which these steps are performed is important because we know that electronic devices like cameras need power in order to function. We apply this knowledge so effortlessly that we are usually not even aware of it. To appreciate the situation from the point of view of a computer, consider the following excerpt from an actual sequence for the Galileo spacecraft, currently in orbit around Jupiter:

```
04284795:56:0 98-001/00:01:00.266 CMD,35A,20ZS3B,PRI,
    98-001/00:01:00.266,1;
04284795:71:0 98-001/00:01:10.266 CMD,35DML,20ZS4A,PRI,
    98-001/00:01:10.266,4000,40,02,C0,46,C4,00,40,00,04,02,
    04,02,04,02,04,02,04;
```

or, translated into English, "At 56 seconds, do command 35A. At 71 seconds, do command 35DML." Unless you happen to be a member of the Galileo sequencing team you probably have no idea what 35A and 35DML actually do, which puts you in the same position as the spacecraft itself. The spacecraft has no knowledge of the interactions between the 35A and 35DML commands, so it has no basis for making any choices about their execution. The spacecraft has no basis for determining whether, for example, the 15 second delay between command 35A and 35DML is really necessary, if it can be longer or shorter, if the order of these two commands could be reversed, and if either of them puts the spacecraft at risk for a bus trip or other catastrophe. All the information needed to make such decisions has been "compiled away" in the process of generating the sequence, so all the spacecraft can do is blindly follow its dictates, and call home if the slightest thing goes wrong.

If we want our spacecraft to exhibit less brittle behavior, then the knowledge that currently gets compiled away during sequence generation must be somehow provided to the spacecraft. One approach is to build an explicit *declarative* model of the spacecraft and use various Artificial Intelligence (AI) search and deduction tecnniques to make operational decisions. Unfortunately, the models needed to support such an approach tend to be very large and difficult to maintain. For example, to figure out from declarative knowledge even as simple a thing as how much timing flexibility exists in the above sequence the spacecraft needs to know what the 35A command does (it turns on the magnetometer), what the 35DML command does (it loads the magenetometer's firmware — DML stands for direct memory load), how much power the magnetometer uses, how long it takes after the magnetometer is turned on before it is ready to accept commands, when the magnetometer is actually needed, how much heat the magnetometer generates, etc. etc. Things get immensely more complicated for commands to, say, turn the spacecraft or fire the main engine.

An alternative to generating a declarative model of the spacecraft (and associated computational machinery) is to incrementally expand the existing *procedural* vocabulary of sequences to include explicit represenations of, say, execution time flexibility. For example, consider the following command "sequence":

Step 1: Do command 35A between times T1 and T2.

Step 2: Do command 35DML no less than 15 seconds after step 1.

It is not possible to give these instructions in a traditional command sequence, where every step must be associated with a particular fixed execution time. With a simple extension to the sequence vocabulary and the underlying execution machinery we have given the spacecraft the ability to display some flexibility in execution without the need to generate a complete declarative model.

We can envision additional increments to the expressiveness of our command language. For example, we might want to protect against the possibility of a bus trip by saying, "Before executing command 35A wait until the system power margin is at least five watts." Here we have extended the vocabulary to to allow the system to wait for a particular condition to become true before taking an action. This naturally leads to a host of other extensions: timeouts, descriptions of corrective action if timeouts expire, etc.

These extensions to the sequencing vocabulary provide the ability to express spacecraft instructions that are not nearly as brittle as traditional sequences. Because they are backwards-compatible extensions to the traditional sequencing paradigm, procedural representations tend to be easier to use than their declarative counterparts, which require a more radical shift in how one thinks about spacecraft commanding. Also, the computational machinery needed to execute procedural representations tends to be relatively simple and efficient.

However, purely procedural representations are limited in that they cannot provide the spacecraft with any knowledge of what the commands it is executing actually *do*. Extending the command vocabulary provides more power to express the *results* of deliberations about command interactions, but we have not done anything to simplify the deliberations themselves. You have to know just as much about the spacecraft to generate a flexible sequence as you do to generate a brittle one.

Fortunately, the declarative and procedural approaches are complementary rather than antagonistic. We have developed a system for executing robust procedures that can serve both as a spacecraft sequencing system, and as an integrating component for a hybrid autonomous control architecture that combines procedural and (multiple) declarative representations. This hybrid architecture, of which our executive is a part, is described in the next section. The executive itself is described in more detail in section 5.

4. Remote Agent Architecture

We have developed an architecture called Remote Agent (RA) [Bernard98] that combines declarative and procedural mechanisms for representing the knowledge required for autonomous operation. (See figure 2.) RA consists of four components: the "smart" executive (EXEC), a mission manager[2] (MM), a

---

[2]The mission manager and planner/scheduler are actually implemented within a single software module called PS/MM (or MM/PS). They use the same modelling language and computational machinery. They are distinguished because they use different models (though the same modelling language) and serve two distinct architectural roles.

planner/scheduler (PS) [Muscettola97], and a system called MIR (mode identification and recovery) [Williams96] for reasoning about the finite-state behavior of the spacecraft. RA will be be used in a flight experiment to control the New Millennium Deep Space 1 spacecraft (DS1) for a one-week period in 1999.

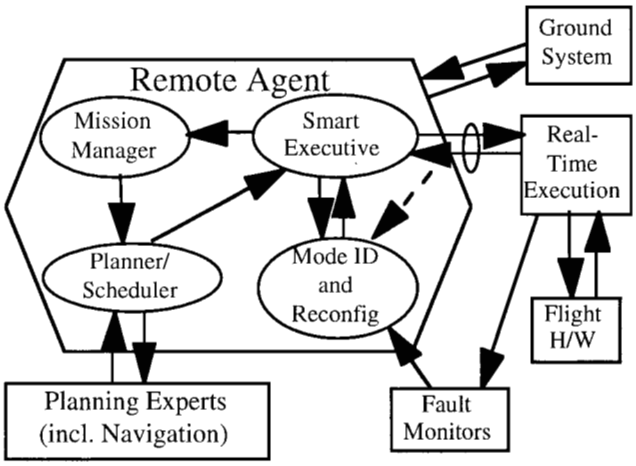

Figure 2: The Remote Agent architecture embedded within flight software

RA interacts with the spacecraft through a *real-time control* system (RT), which directly controls the spacecraft hardware. RT consists of a set of control and monitoring tasks which provide the core functionality for the RA. RT functionality ranges from relatively simple device drivers that provide direct access to spacecraft hardware, to complex control loops for controlling the spacecraft's attitude and trajectory. Information about the status of RT control loops and hardware sensors is passed back to RA either directly or through a set of *monitors*. The PS component of RA also interacts with a set of planning experts, which are subroutines for computing answers to domain-specific queries like how much time it will take to turn from one attitude to another.

EXEC is the core of the architecture. It coordinates all the activities in both RA and in the external software components. EXEC provides both an extended procedural vocabulary for expressing instructions, and an integrated declarative knowledge base. EXEC itself has very limited deduction capabilities, relying instead on MM, PS and MIR to perform most of the model-based reasoning.

PS and MIR are both model-based, and each has its own modelling language. The PS modelling language (which is also used for MM) has facilities for describing the interactions of spacecraft activities evolving in time, and allows one to describe constraints among acitivities (like turns) and states (like hardware configurations). The language also has vocabulary for talking about time which is used to describe how long different activites are expected to take. PS uses this information to generate plans of action in terms of activities bound to temporal intervals whose endpoints are not fixed in time, but simply constrained relative to one another. The resulting plans are thus

both concurrent and flexible, that is, they describe activities that go on in parallel, and without specifying precisely when those activities are to take place.

The MIR modelling language has facilities for describing the finite-state behavior of the spacecraft hardware. For example, using the MIR modelling language one can express the fact that if the power switch for a device is closed, and enough power is available, that the corresponding device should be on. The MIR models are used in two ways, for fault diagnosis (mode identification or MI) and fault recovery (mode recovery or MR). Using its models, MIR is able to deduce the most likely actual state of the spacecraft hardware given the observable state, i.e. sensor readings, taking into account that the sensors themselves might be faulty. MIR can also use the same models to derive sequences of actions that will produce a desired configuration of the hardware. These capabilities are used to provide EXEC with an abstracted view of the spacecraft state, and a mechanism for recovering from faults.

RA is written entirely in Common Lisp. It runs both in Allegro Common Lisp under Unix, and on the flight processor (a Rad6000 - a radiation-hardened processor similar to a PowerPC) under a custom port of Harlequin Common Lisp for the vxWorks real-time operating system.

5. Remote Agent Executive

The Remote Agent Executive coordinates the activities of all the other components of the spacecraft software. EXEC consists of a core language which provides an expressive vocabulary for procedural knowledge, and a set of higher-level facilities built on top of this core, including flexible plan execution, configuration management, and resource management. Each of these facilities is described in turn in the following subsections.

5.1 ESL

The core of EXEC is a lanaguage called ESL (Execution Support Language) [Gat96] which provides a rich representation for the sort of procedural knowledge alluded to in section 3 above. ESL consists of about half a dozen loosely coupled feature sets that provide facilities for handling unexpected contingencies, achieving goals, managing and coordinating parallel tasks, and managing resources. In addition, ESL provides a backchaining logical database, essentially a little Prolog interpreter, which is the central mechanism through which the procedural and declarative paradigms interact.

5.2 Flexible Plan Execution

EXEC provides a facility for executing the plans produced by PS, which are not linear sequences of commands, but rather abstract descriptions of parallel activities. A PS plan consists of a set of timelines. Each timeline corresponds to a description of some aspect of the spacecraft's state at varying levels of abstraction. For example, a timeline may describe a simple state like whether a device is on or off, or a more complex abstract state like whether or not the spacecraft is turning, and what its target attitude is.

Each timeline is subdivided into a set of temporal intervals over which the timeline's state description is constant. These intervals are referred to as *tokens*. For example, the timeline for the power state of a device usually

consists of a sequence of tokens alternating between *on* and *off*. The timeline for the spacecraft attitude will alternate between *turning* and *constant-attitude* tokens.

The beginning and ending times of the tokens are in general not specified in terms of absolute time, but rather in relation to the start and end times of other tokens. This allows an explicit representation of the dependencies among tokens, and enables considerable robustness and flexibility. For example, consider the example from section 3, where the magnetometer had to be turned on fifteen seconds before loading its software. This dependency can be represented in a PS plan by two tokens, one on the magnetometer's power state timeline which states that the power is on, and another on the magnetomer's activity timeline which states that the software is being loaded. The temporal constraint is represented by a plan annotation that says that the start of the first token must come at least fifteen seconds before the start of the second token.

As one might imagine, executing a plan specified using this notation is considerably more challenging than executing a simple linear sequence of commands. But it generates behavior that is far more flexible and reliable because the executive now has explicit information about the dependencies between steps. In a traditional sequence, if anything went wrong while turning on the magnetometer the entire sequence would fail because the spacecraft would have no way to know the potential repurcussions of this deviation from the plan. With the new representation, EXEC knows exactly what the dependencies are. If turning on the magnetometer fails, EXEC is free to take some corrective action as long as the software-loading activity doesn't start for at least fifteen seconds after the power is successfully turned on, and all the other temporal constraints specified in the plan are met.

EXEC's plan execution system (called the plan runner) is built on top of ESL's task management facilities. Each plan timeline has a thread of execution associated with it. Each thread issues commands to control those aspects of the spacecraft state corresponding to its timeline. The threads use ESL's synchronization facilities to coordinate their actions to conform to the temporal constraints specified in the plan. The result is a system that is far more robust than a traditional sequence-based executive.

5.3 Resource Management

Scarce resources are a fact of life on spacecraft. On-board processes contend for a limited number of devices. Shared resources, such as energy and data storage, are subject to hard (and relatively severe) limits, as well as environmental influences (such as solar exposure for battery charging) that change these limits over time.

Resource management in RA is done mainly by PS, which typically uses worst-case estimates of resource utilization. However, the uncertainty in the environment and its effect on the true resource utilization can make the PS resource usage estimates inaccurate. The problem is particularly severe on planetary rovers, which must contend with huge environmental uncertainties.

EXEC provides a general architecture for managing resources at run-time, providing a layer of protection against inaccurate predictions by PS, and also a measure of safety for manually generated procedures. EXEC's resource

manager (RM) architecture consists of a managing object for each resource. When a task wants to take an action that affects the state of a resource it registers its intent to do so with the resource manager object. The manager object checks all the intentions pertaining to a particular resource and finds a mutually consistent set. Those tasks making the mutually consistent requests are allowed to proceed; all other requests are blocked until enough tasks in the first set release their requests to allow a new mutually consistent set to proceed.

This general architecture is currently implemented only for discrete-state resources like hardware configurations. The manager object for a discrete state resource is called a *property lock*. Property locks and the facilities for accessing them are currently built in to ESL.

The property lock mechanism is quite powerful. It automatically manages not only the synchronization of multiple tasks accessing a particular resource, but also the actual achievement, maintenance and (if needed) restoration of the physical state corresponding to that resource. However, it also lacks some important functionality. It does not take into account process priorities. (This is a very complicated problem; see section 6.) It does not currently perform deadlock detection, although this could be added without difficulty. And, as mentioned before, the property lock mechanism is limited to managing discrete states.

5.4 Configuration Management

PS plans describe the evolution of the spacecraft state usually at a high level of abstraction, leaving it up to EXEC to fill in the details at run time. For example, the plan may specify that a particular capability is needed for a particular activity, but it might not say anything at all about the actual hardware configuration required to provide that capability. It is the responsibility of EXEC to manage the actual configuration of the spacecraft to make it conform to the constraints imposed by the plan. This is challenging for several reasons. First, spacecraft are quite complicated, with many tightly coupled interacting components. (As discussed in section 2, even as simple an action as turning on a device can be quite complicated.) The best configuration for a particular goal is often a function of the current configuration. Second, the design of the spacecraft can change during the software development process. Third, the spacecraft is expected to continue to operate even if some of its components fail in flight.

EXEC provides a generic configuration management facility (CM) which uses a simple model of the spacecraft hardware topology to automatically generate the right sequence of commands to control the spacecraft configuration. The model is described in terms of a class hierarchy and a connectivity diagram. A generic library of configuration management routines uses the model to control the hardware.

A typical model includes descriptions of all the devices, their types, and their data and power connections. CM uses this information to respond to requests to provide devices of a particular class. For example, if a request is made to provide a camera, CM will go through the list of available cameras, select one, and configure it. This typically involves considerations of hardware states other than just the camera. For example, turning on a camera usually involves sending a command to a power switching unit, not the camera.

Besides configuring the hardware to provide requested functionality, CM also coordinates configuration requests from multiple parallel tasks at run-time using ESL's property lock mechanism. This provides an additional measure of safety against flaws in the PS plan. (PS plans are guaranteed to be correct with respect to the PS models, but the models can have errors with respect to the actual situation on the spacecraft.)

CM makes hardware configuration management code much easier to write and maintain than hand-crafted code. If the spacecraft configuration changes, only the hardware model needs to be changed. Commanding a particular configuration usually involves only one line of code.

For example, the following code is an excerpt from the DS1 CM model.

```
(define-device-class :camera
  :power-function fsc-power-request
  :talk-function camera-talk-msg)

(define-device :camera_A :camera
  :powered-thru :power_bus_1
  :switched-thru :fsc_camera_switch1
  :ready-state ((:health_state :ok)
                (:power_state :on)))
```

Based on the above model, configuring and using a camera requires only a tiny snippet of code:

```
(with-selected-device :camera
  (take-pictures))
```

This code would select a camera, make it ready by taking actions to make it powered on and healthy, and then take pictures. Until the picture-taking is complete, any task that tries to change the state of the camera away from its ready state will be prevented from running. Furthermore, if the camera deviates from its ready state during picture taking then two things will happen. First, the picture-taking task will be notified that the camera is no longer ready and given an opportunity to either 1) take corrective action or 2) wait for an automatic recovery (generated by MIR). All this complexity is completely hidden from the programmer behind the abstractions of high-level control constructs like WITH-SELECTED-DEVICE. More information on how contingencies are handled can be found in [Gat97].

6. Related Work

There are many robust execution systems in the literature. The one most closely related to EXEC is RAPs [Firby89] and the related 3T architecture [Bonasso97]. In fact, an early version of EXEC was implemented using RAPs. Unlike ESL, RAPs is not an extension to a general-purpose programming language, but a unique language in its own right with its own semantics. This made it cumbersome to combine robust execution with general-purpose computation, which is what motivated the development of ESL. RPL [McDermott91] is another direct decendent of RAPs whose design influenced ESL, along with PRS [Georgeff87] and RL [Lyons93]. [Freed98] describes a RAP-like language that was used in an interesting automated air-traffic control application.

At the other end of the implemetation spectrum, TCA [Simmons90] is a robust execution system that is implemented not as a language or a language extension, but as a subroutine library. TCA combines a robust execution system with an interprocess communications system, and uses message passing to excercize control over processes. Because TCA does not provide new control constructs, it requires quite a bit of dicipline on the part of the programmer to hand-compile his intentions into TCA calls. A recent refinement of TCA embeds the subroutine library within an extension of C++ called TDL [Simmons98].

7. Summary and Future Work

Traditional spacecraft commanding requires ground operators to generate detailed and inflexible command sequences. These sequences are difficult and expensive to generate, and do not contain enough information to allow the spacecraft to respond intelligently to deviations from expected behavior, which makes them inherently brittle.

At the core of a new control architecture (the Remote Agent) designed to make spacecraft more self-reliant and less dependent on ground intervention is an executive (EXEC) which implements an expanded vocabulary for commanding spacecraft. EXEC provides a wide range of capabilities at different levels of abstraction, ranging from simple failure recovery mechanisms, to the execution of high-level flexible plans and the ability to command the spacecraft directly in terms of abstract hardware configurations. EXEC's design is highly modular, making it able to integrate a wide variety of external facilities. In the RA architecture, EXEC integrates a planner and scheduler (PS), a discrete-state reasoning system (MIR), and real-time control software (RT).

EXEC's flexibility gives it much of its power, but it is also a source of certain weaknesses. Because there are few architectural requirements levied on the modules that EXEC connects, the result can be a mishmash of different conflicting representations. RA uses three different representation languages. It represents redundant information in all three, which can cause serious software configuration control problems.

EXEC's resource management capabilties are also somewhat impoverished at the moment. It can only manage discrete-state resources, it does not take task priority into account when making resource decisions, and it does not have a mechanism for detecting and recovering from deadlocks. However, the general architecture was designed to allow all these capabilities to be added without major difficulty.

In the future we hope to develop an autonomy system that uses a more uniform and non-redundant representation among its different components. We also intend to remedy the three shortcomings of the resource manager listed above. Taking task priorities into account presents a unique problem. In general, high-priority tasks should be allowed to usurp resources from low-priority tasks. However, this leads to very difficult problems in trying to insure the consistency of the dynamic state of the usurped task. This is because, in general, a task may have associated with it a number of *cleanup procedures* that must be executed before the task can be safely aborted. But these cleanup procedures can, in general, take arbitrarily long to execute. In

general, this problem devolves to the full planning problem. The challenge will be to find a heuristic that handles a useful subset of the general case.

Acknowledgements

This work was carried out partly at the NASA Ames Research Center and partly at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the NASA. We gratefully acknowledge the contributions of the EXEC team members past and present, including Gregory Dorais, Charles Fry, Edward B. Gamble, Jr., Ron Keesing, Christian Plaunt, and Richard Washington.

References

[Agre90] Phillip Agre and David Chapman. "What are Plans For?" *Robotics and Autonomous Systems*, vol. 6, pp. 17-34, 1990.

[Bernard98] Douglas E. Bernard, Gregory A. Dorais, Chuck Fry, Edward B. Gamble Jr., Robert Kanefsky, James Kurien, William Millar, Nicola Muscettola, P. Pandurang Nayak, Barney Pell, Kanna Rajan, Nicolas Rouquette, Benjamin Smith, Brian C. Williams, "Design of the Remote Agent Experiment for Spacecraft Autonomy", in *Proc. of IEEE Aeronautics Conference (Aero-98)*, Aspen, CO, IEEE Press, 1998.

[Bonasso et al. 97] R. Peter Bonasso, et al. Experiences with an Architecture for Intelligent Reactive Agents. *Journal of Experimental and Theoretical AI*, 9(2), 1997.

[Firby89] R. James Firby. Adaptive Execution in Dynamic Domains, Ph.D. thesis, Yale University Department of Computer Science, 1989.

[Freed98] Michael Freed. "Managing Multiple Tasks in Complex Dynamic Environments." AAAI98.

[Gat97] Erann Gat, "ESL: A language for supporting robust plan execution in embedded autonomous agents," in Proc. of IEEE Aeronautics (AERO-98), Aspen, CO, IEEE Press, 1997.

[Gat98] Erann Gat and Barney Pell, "Abstract Resource Management in an Unconstrained Plan Execution System," in Proc. of IEEE Aeronautics (AERO-98), Aspen, CO, IEEE Press, 1998.

[Georgeff87] Michael Georgeff and Amy Lanskey, "Reactive Reasoning and Planning", *Proceedings of AAAI-87.*

[Lyons93] Damian Lyons. "Representing and Analyzing action plans as networks of concurrent processes, " *IEEE Transactions on Robotics and Automation*, 9(3), June 1993.

[McDermott91] Drew McDermott. "A Reactive Plan Language," Technical Report 864, Yale University Department of Computer Science, 1991.

[Muscettola97] N. Muscettola, B. Smith, S. Chien, C. Fry, G. Rabideau, K. Rajan, D. Yan, "On-board Planning for Autonomous Spacecraft," in Proceedings of the fourth International Symposium on Artificial Intelligence, Robotics and Automation for Space (i-SAIRAS 97), July 1997.

[Pell96] B. Pell, D. E. Bernard, S. A. Chien, E. Gat, N. Muscettola, P. Nayak, M. D. Wagner, and B. C. Williams, "A Remote Agent Prototype for Spacecraft Autonomy," SPIE Proceedings Volume 2810, Denver, CO, 1996.

[Pell97] Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith, "Robust Periodic Planning and Execution for Autonomous Spacecraft," In Procs. of IJCAI-97, 1997.

[Pell98a] Barney Pell, Ed Gamble, Erann Gat, Ron Keesing, Jim Kurien, Bill Millar, P. Pandurang Nayak, Christian Plaunt, and Brian Williams, "A hybrid procedural/deductive executive for autonomous spacecraft," In M. Wooldridge and K. Sycara, eds, Procs. of Second International Conference on Autonomous Agents, AAAI Press, 1998.

[Pell98b] Barney Pell, Scott Sawyer, Douglas E. Bernard, Nicola Muscettola, and Ben Smith, "Mission Operations with an Autonomous Agent," In *Proc. of IEEE Aeronautics Conference (Aero-98)*, Aspen, CO, IEEE Press, 1998.

[Simmons90] Reid Simmons. "An Architecture for Coordinating Planning, Sensing and Action." *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Simmons98] Reid Simmons, "A Task Description Language for Robot Control." Unpublished manuscript.

[Williams96] B. C. Williams and P. Nayak, "A Model-based Approach to Reactive Self-Configuring Systems," Proceedings of AAAI-96, 1996.